Nix and Open Source Software for the Arts

Abstract — Functional programming is a subset of programming based upon Lambda calculus, it treats programming in a mathematical sense where the fundamental building block of the program are functions and data is immutable. Nix is a purely functional package manager used to create reproducible binaries and development environments, it treats packages, and program dependencies as immutable values that are built by pure functions and controlled by a lock file. Nix leverages **nixpkgs**, an open source collection of over 120,000 software packages that can be installed with the Nix, making its GitHub repository one of the largest and most active in the world. In this work we will be reviewing nixpkgs and how open source software relates to the arts, as both a means software preservation and distribution, as well as evaluating the technological sustainability of such open source projects by examining the licenses used for packages in the nixpkgs repository.

I. An introduction to functional programming

Functional programming is a programming paradigm, whose basal unit of computation is a function, a self contained module of code that accomplishes a specific task [1]). Complex programs are then built through the composition and chaining of these functions. There are a few core tenets to the functional paradigm, the main two being **immutability** and **functional purity**, both of which are intrinsically linked. The concept of functional purity is that given the same inputs, a given function will always produce the same outputs. Pure functions are said to have referential transparency. A classic example of this would be a function which given any number n, adds one to it and returns the result: $f(x) = x + 1$. This function, when given the input 3 always produces 4. In a programming language such as Haskell, it would be written as:

```haskell
addOne :: Int -> Int
addOne x = x + 1
```

Figure 1 - A pure function which always adds one to a number in Haskell. As Haskell enforces type level functional purity, impure functions always have an `IO type`. An impure version of the function would have the type `addOne :: Int -> IO Int`.

However, software is rarely pure, that is to say, that there are many conditions in which the output of a function can change. For instance if a function requires a connection to a database or to read a file then connection can fail and the file may not be present. Both of these conditions are elements outside our code which can have an affect on the results a function which relies on said database or file. This ambiguity is referred to as a side effect, and functions that perform them are deemed impure. In purely functional languages, even mutating a variable is considered a side effect, as calling a function which uses said mutable variable as an argument multiple times may yield different outputs.

Referential Transparency The ability to replace a function with its results without affecting the surrounding program.

The package manager and build system, Nix, is purely functional and operates under this same functional programming maxim of immutability. In Nix the packages used to build development environments and software are immutable, and never change after being built. Nix stores packages in a Nix store, where each package has its own unique directory denoted by a cryptographic hash of the package's build dependency graph. Nix (the package manager) configured in Nix (the functional programming language), where packages are built from *Nix expressions*. A Nix expression describes all aspects of a package build action to form a derivation. Nix expressions are deterministic, as in that building a package from an Nix expression twice will produce the same output.

Reproducibility Reproducibility is a property of a computation satisfying the following condition:

$$\forall c \in C, \forall i \in I, \forall e_1, e_2 \in E, \text{eval} = (c, i, e_1) = \text{eval} = (c, i, e_2)$$

where

- C is the set of all possible computations
- I is the set of all possible input arguments
- E is the set of all possible execution environments (such as hardware, software, spacetime)

If a computation satisfies this condition then it said to be reproducible [2].

II. Nix the functional programming language

Nix a domain specific purely functional, dynamically type and lazily evaluated programming language [3]. Nix expressions are expressions written in the Nix programming language, a language with a JSON-esque syntax. Nix even describes itself as JSON with functions. Values in the Nix programming language can be either primitive data type, lists, attribute sets and functions [4]. An attribute set in Nix is a collection of name value pairs, similar to dictionaries or hash maps in other languages.

```nix
{
  string = "hello";
  integer = 1;
  float = 3.141;
  bool = true;
  null = null;
  list = [ 1 "two" false ];
  attribute-set = {
    a = "hello";
    b = 2;
    c = 2.718;
    d = false;
  }; # comments are supported
}
```

Figure 2. - An attribute set in the Nix programming language.

Attribute sets can also inherit from the surrounding lexical scope of a Nix expression and either other attribute sets allowing for greater flexibility[5]. Here, `x: inherit x; y = 80` is a function which returns an attribute set in which the x value is inherited from the x function argument. Being a functional programming language, functions are at the core of Nix code. In Nix functions always take exactly one argument and the argument and function body are separated by a `:`, where the left side is the function's argument and the right the body. All functions in Nix are lambda expressions.

```
{ a, b }: a + b
```

Figure 3. - A Nix function that takes an attribute set as an argument.

Build inputs in Nix are specified explicitly in two ways, file system paths or through dedicated functions. In Nix, whenever a system path is used via string interpolation, the contents of the file in that path are copied to Nix's own filesystem abstraction, the **nix store** [6]. In Nix, build inputs do not just have to come from the local file system, as part of the Nix's standard library, a verity of built in impure functions are provided to fetch files over the network, namely:

- `builtins.fetchurl`
- `builtins.fetchTarball`
- `builtins.fetchGit`
- `builtins.fetchClosure`

Naturally an error occurs if the network request fails, or the if the file is not present at its specified location when using local inputs.

Dynamic Typing Where a programming language checks its variables type at runtime based on its assigned value.
Lazy Evaluation Where expressions in a program are note evaluated until their values are needed.
Build Inputs Build inputs are files that derivations refer to in order to describe how to derive new files. When run, a derivation will only have access to explicitly declared build inputs.

III. Nix the build system

Build systems automate the execution of repeatable tasks for for users. The goal of a build system is to build up an up to date store, which maps keys to their associated value. In software, the store is projects filesystem and keys are the files and the values are the file contents [7]. A build system takes a *task*, a target key and store and returns a new store in which target key and all its values have an up to date value, this kind of action, usually referred to as a build step can be seen a function with the type signature: `build :: Tasks c k v-> k-> Store i k v-> Store i k v`. Most build systems also contain some form of persistent build information, which retains information between invocations of the build system. Build systems are composed of a `scheduler` and a `rebuilder`.

Task A specification on how compute a new value from a key a in a build system. Requires user input.

A `scheduler` is a function which takes two arguments, a rebuilder and constructs a build system but determining the order in which keys must be rebuild. There are a few different kinds of schedulers, namely:

1. **Topological** — This kind of scheduler computes build tasks in a linear order, it does so by finding an acyclic graph of the each key's reachable dependences then computing a topological sort.
2. **Restarting** — Restarting schedulers execute build tasks in an arbitrary order, evaluating their dependencies on the fly and whenever a task calls an out of date dependency, *abort* the task and rebuild that said dependency. The discovered task order and the most recent dependency information is often recorded as a persistent build information, often reducing the number of task aborts needed in a build to near 0.
3. **Suspending** — A suspending scheduler builds dependencies as they are requested, simply suspending the current task on such occurrences. Suspension of tasks can be done via cheap green threads or continuation passing style [8].

A `rebuilder` is a function which takes three arguments, a key (file), a value (file contents) and a task which is used to recompute the value of the key if deemed necessary. However, if it is unnecessary to update a keys values then the `rebuilder` function simply returns the value passed in as an argument unchanged. There are a few common strategies for rebuilding.

1. **Dirty Bit** — A piece of persistent information for all keys in a build system, a `Bool` value indicating whether a key is dirty (it's value has changes) or clean (it's value is unchanged since last build). If a key and all its transitive dependencies are clean then the key does not need to be rebuild. Every build all keys bit are set to clean.
2. **Verifying Traces** — A record of a keys values or hashes of dependencies used in the last build, if something has changed then that key is dirty. To verify if a key needs to be rebuilt a function which takes a target key, a hash of the key's current value and and a function for getting the post build value of any key in the system (this is done in conjunction with that build systems scheduler). The result of this function is a `Bool` where `True` indicates that the key does not need rebuilding and `False` indicates the opposite.
3. **Constructive Traces** — Whereas verifying traces rely on hashing to keep a compact profile, constructive traces record the full previous value of a key rather than just its hash, storing complete results. Multiple traces may be associated with a single key, and this rebuild strategy is particularly valuable in cloud-based systems where sharing and reusing full outputs improves efficiency.
4. **Deep Constructive Traces** — Constructive traces always verify key by looking at it's intermediate dependencies, which require each of these transitive dependencies to be first brought up to date, making the time it takes to verify a key reliant on its number of intermediate dependencies. A Deep constructive trace optimises this process by only looking at the terminal input keys, ignoring any intermediate dependencies [7].

When needed both the `scheduler` and `rebuilder` functions may use persistent build information to augment their processes. Build systems are designed to be both correct and fast, achieving this largely through the use of robust dependency graphs defining the inputs for every build target in the project (source files, configuration, tools) and clever hashing of file contents such that even if a files time stamp has changed, if the file remains genuinely the same, then re-compilation does not need to occur for that file, greatly reducing compile times. This is referred to as Minimality.

Stale binaries A compiled program or component file which is out of date with respect to the rest of the source code or the dependencies which is built upon. Stale binaries can lead to incorrect behaviour and hard to trace bugs in a codebase.
Minimality A build system is minimal if it executes tasks at most once per build, and only if they transitively depend on inputs that changed since the previous build [7]

Nix is as much a build system as it is a package manager. It uses a *suspending scheduler* and *deep constructive trace rebuilder.* Nix can also distribute the software that it builds as a package. A Nix package is an expression in the Nix programming language which will evaluate to a derivation, a sequence of build steps needed to generate the data required to make a given piece of software. Nix expressions describe how to build packages from source and collected in the *nixpkgs* repository. Using these expressions the nix package manger can build binary packages, as well a development environments known as Nix-shells. Nix has uses its own immutable abstraction of the file system known as a Nix store. The default nix store on the local files systems is /nix/store. They are a collection of store objects, a store object consists of:

- A file system object as data
- A set of store paths as references to store objects

A File system object will be one of the following:

- A file
- A Directory
- A Symbolic Link (A file which refers to another file in another directory as a path)

A file system object can be either a file/directory or symbolic link (A file which refers to another file in another directory as a path). A store path is an opaque unique identifier. A store path will always reference exactly one store object. Store objects are pairs of:

1. A 20-byte digest for identification
2. A human readable symbolic name

Store objects are immutable, as in that once they are created, they do not change nor can any other store object that they reference can be changed. The store plays a pivotal role in way Nix creates derivations for building software, as derivations themselves are specifications for running a given executable on the specified input.

Opaque Datatypes A datatype with an internal structure that is not available for inspection.

Nix Derivations A specification for running an executable on precisely defined input to produce on more store objects. These store objects are known as the derivation's *outputs.* Derivations are *built*, in which case the process is spawned according to the spec, and when it exits, required to leave behind files which will (after post-processing) become the outputs of the derivation.

Naturally Nix derivations can be made in conjunction with language specific package managers such as Rust's `cargo`. With many open source projects such as *poetry2nix*, *uv2nix* and *cabal2nix* capable of creating Nix derivations based upon their languages respective package managers. For languages such as Rust, Nix offers first class support for making derivations based upon a rust projects `Cargo.toml` and `Cargo.lock` files.

```
manifest = (pkgs-stable.lib.importTOML ./
Cargo.toml).package;

rustPackage = pkgs-
stable.rustPlatform.buildRustPackage {
    inherit (manifest) name version description;
    cargoLock.lockFile = ./Cargo.lock;
    src = pkgs-stable.lib.cleanSource ./.;
};
```

Figure 4. - Parsing Rust `Cargo.toml` with the Nix function `pkgs-stable.lib.importTOML` and then reading the rust source code dependencies using the function `pkgs-stable.rustPlatform.buildRustPackage`. Both of these functions are part of Nix's standard library.

The `rustPlatform.buildRustPackage` function takes the source code from `src` and the projects `Cargo.lock` file and builds the project as via `Cargo` using the Rust toolchain already available no nixpkgs.

IV. Nix the package manager

A package manager is a tool that automates code reuse. They ensuring that all third party dependencies needed for a project, either directly or indirectly are downloaded and version controlled in manner which no dependencies conflict [9]. Package managers usually operate at two levels, either tied to a programming language like the package manager `cabal` is tied to Haskell, where they are tasked with finding dependencies related to a projects source code, or at a system level managing full software installation such as `brew` on MacOS.

The Nix package manager operates on the system level, and leverages *nixpkgs*, a software distribution built with Nix (the programming language), it contains 120,000 packages all released under the various open source licenses. These packages can then be used with Nix on most Linux distributions. Packages are distributed via channels. For use on non NixOs Linux distributions, package distribution is done through the `nixpkgs-unstable` channel. Both `nixos-unstable` and `nixpkgs-stable` follow the master branch of the nixpkgs repository.

Nix Channels A pointer to versioned tarballs and git commits.

Nix supports a variety of platforms with various amounts of support for each:

1. `x86_64-linux`: Highest level of support.
2. `aarch64-linux`: Well supported, with most packages building successfully in CI.
3. `aarch64-darwin`: Receives better support than `x86_64-darwin`.
4. `x86_64-darwin`: Receives some support.

Nix packages can be modified and extended via *Overlays.* Overlays are Nix functions which accept two arguments (idiomatically referred to as `final` and `prev`) and return a set of packages. They are similar to the `packageOverrides` function, but more flexible, as they are capable of taking in arguments beyond `final` and `prev`. Overlays are commonly used to either patch, or pin package to a version that nixpkgs does not ship. An example of which is the following:

```
final: prev:
{
    sl = prev.sl.overrideAttrs (old: {
        src = prev.fetchFromGitHub {
            owner = "mtoyoda";
            repo  = "sl";
            rev   =
"923e7d7ebc5c1f009755bdeb789ac25658ccce03";
            hash  =
"173gxk0ymiw94glyjzjizp8bv8g72gwkjhacigdlan09jshdrjb4";
        };
    });
}
```

Figure 5 - Example Nix expression pinning the `sl` package specifically to the `923e7d7ebc5c1f009755bdeb789ac25658ccce03` revision. Placeholder hashes such as `sha256-AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA=` can be used, and Nix's `hash mismatch error` will provide the correct one as part of its error message upon evaluation of the Nix expression.

It is similarly easy to apply a patch to package in Nix.

```
final: prev:
{
  sl = prev.sl.overrideAttrs (old: {
    patches = (old.patches or []) ++ [
      (prev.fetchpatch {
        url = "https://github.com/charlieLehman/sl/
commit/e20abbd7e1ee26af53f34451a8f7ad79b27a4c0a.
patch";
        hash =
"07sx98d422589gxr8wflfpkdd0k44kbagxl3b51i56ky2wfix7rc";
      })
      # alternatively if you have a local patch,
      /path/to/file.patch
      # or a relative path (relative to the current
nix file)
      ./relative.patch
    ];
  });
```

Figure 6. - Patching a package in Nix. This is done by overriding the source `src` of the target package with the patched version [10]

By building software as a Nix package, Nix also becomes a means for software distribution. The command `nix run` allows for the execution of applications or binaries provided by Nix packages or flakes. Making it a streamlined way to run software without installing it system-wide. The command: `nix run blender-bin`. Runs the default app from the `blender-bin` flake. To use `nix run` the flake must evaluate to an app or regular Nix derivation. If it evaluates to an app, then `nix run` executes the program specified by the app definition, by building it from source, it does not download the app specified by the derivation, allowing for the temporary use of applications.

```
apps.x86_64-linux.blender_2_79 = {
    type    = "app";
    program = "${self.packages.x86_64-
linux.blender_2_79}/bin/blender";
};
```

Figure 7. - A Nix app definition. Here the target system is specified with `apps.x86_64-linux`

If `nix run` evaluates to a derivation, then Nix will try to execute the program `<out>/bin/<name>`, in which `out` refers to the primary output store path of the derivation and `name` refers to either:

- The `meta.mainProgram` attribute of the derivation.
- The `pname` attribute of the derivation.
- The `name` attribute of the derivation.

V. Nix Flakes

Flakes are the unit for packaging Nix code in a reproducible and discoverable way. A flake is a filesystem tree often fetched in the form of a git repository or tarball, which contains `flake.nix` file in the filetree's root. `flake.nix` specifies some metadata about the flake such as inputs (the flakes dependencies) and its outputs (Nix values such packages or modules for use in NixOs systems). They can have dependencies on other flakes, making it possible to have multi-repository Nix projects. They have increasingly become the standard for writing and distributing Nix derivations within the community. All Nix flakes share a common anatomy, they begin with an optional description of the flake and it's declared inputs, which are the dependencies for the flake.

```
description = "A very basic Haskell dev env flake";
inputs = {
    nixpkgs-stable.url   = "github:NixOS/nixpkgs";
    nixpkgs-unstable.url = "github:NixOS/nixpkgs?
ref=nixos-unstable";
};
```

Figure 8. - Input syntax for a Nix flake. Nix is not space sensitive allowing for code alignment for better readabilty.

Flake inputs can be either local or remote, with a differing syntax for each. Remote inputs have a url-like syntax such as: `github:NixOS/nixpkgs`. If the flake input is refers to a local repository, then it will take the form of `git:/home/user/sub/dir`. If the path of the flake input begins with `./` or `/` then it is treated as local path. Inputs in a Nix flake can **inherit** from other inputs, which is useful in minimizing flake dependencies, and can simplify any potential debugging. Input inheritance has the following syntax: `inputs.nixpkgs.follows = "dwarffs/nixpkgs";`. Inputs in a `flake.nix` tend to not have the specific version hashes referenced within the input URL. Instead to ensure reproducibility Nix generates a `flake.lock` file in the flakes directory on the first evaluation of said flake. The lockfile is a UTF-8 JSON file containing the graph structure isomorphic to the graph of dependencies for the root flake. An example flake.lock file is would the following:

```
{
  "nodes": {
    "nixpkgs-stable": {
      "locked": {
        "lastModified": 1765075010,
        "narHash":
"sha256-8pbe+pDKWWy0qZ4KSz+OtWm2xuLZp4ubPJDuc0TIHi4=",
        "owner": "NixOS",
        "repo": "nixpkgs",
        "rev":
"b12293a0adf2e81e6f6164edd4f4cc8d4e5c17d4",
        "type": "github"
      },
      "original": {
        "owner": "NixOS",
        "repo": "nixpkgs",
        "type": "github"
      }
    },
    "nixpkgs-unstable": {
      "locked": {
        "lastModified": 1764950072,
        "narHash": "sha256-
BmPWzogsG2GsXZtlT+MTcAWeDK5hkbGRZTeZNW42fwA=",
        "owner": "NixOS",
        "repo": "nixpkgs",
        "rev":
"f61125a668a320878494449750330ca58b78c557",
        "type": "github"
      },
      "original": {
        "owner": "NixOS",
        "ref": "nixos-unstable",
        "repo": "nixpkgs",
        "type": "github"
      }
    },
    "root": {
```

```
      "inputs": {
        "nixpkgs-stable": "nixpkgs-stable",
        "nixpkgs-unstable": "nixpkgs-unstable"
      }
    }
  },
  "root": "root",
  "version": 7
}
```

Figure 8. - An example `Flake.lock` file, containing 2 nodes and the root flake.

A resolved `Flake.lock` file dependency graph structure is represented through nodes, each node will contain:

- `inputs` - The dependencies of this node, as a mapping from input names (e.g. `nixpkgs`) to node labels.
- `original` -The original input specification from `flake.nix`, as a set of `builtins.fetchTree` arguments.
- `locked` - A set of `builtins.fetchTree` arguments mapped to a specific revision denoted by the hash of the `rev` field.
- `flake` - A Boolean denoting whether this is a flake or non-flake dependency. Corresponds to the `flake` attribute in the `inputs` attribute in `flake.nix`.

Lock files are the key component of reproducible builds and remain static, never changing once resolved. The only way to update the inputs in a nix flake is either manually by editing that inputs' associated revision and hash and changing them to desired version or through the command `nix flake lock --update-input <input-name>` which will update the specified input, `<input-name>`. It is also possible to update all inputs in a flake via the command `nix flake update`, which will generate an entirely new `flake.lock` in-place. If a flake's lock file is deleted, a new one will be generated upon the next evaluation of the flake and dependency versions may differ.

Lockfile A type of file generated usually generated by a package manager, it is typically not meant to be edited, but is instead a reflection of the resolved dependency tree for a project [11]. They usually have the file type `.lock`.

Once the inputs of a flake are resolved, they are passed to the function which returns the output of flake. This function always takes in `self` and the input variables as arguments. Flakes in Nix can output source code as packages, development environments, and even the outputs of other flakes.

```
outputs = { self, nixpkgs }: {
    packages.x86_64-linux = {
        default = self.packages.x86_64-linux.hello;
        hello = nixpkgs.legacyPackages.x86_64-
linux.hello;
    };
};
```

Figure 9. - A very simple flake output which exposes the hello package for the `x86_64-linux` architecture.

VI. Nix Shells

Setting up complex development environments is incredibly time consuming, and only grows in complexity when taking to account the different configurations individual systems will have. Both Nix and Docker take different approaches to solving this problem. Docker is a containerisation platform. Containerization allows for applications and their dependencies to be isolated. Each container contains everything need for the application to run, including the application's code/binary, its required libraries and configuration file [12]. Docker containers all share the kernel on the host machine, but isolate the filesystem, networking and application processes. Nix can set up a shell environment called a `nix-shell`, it is a much more lightweight approach to reproducible environments that Docker. To load a shell environment, Nix first evaluates the `shell.nix` or `flake.nix` file in the root of the current working directory add loads a shell with the specified packages at their locked versions on path. As apposed to Docker, the package dependencies are entirely reproducible. Like Docker, nix-shells do share the host machine kernel but it does not isolate beyond package versions. nix-shells do not permanently install a package, as once you exit a shell, that shell's packages will no longer be available.

```
devShells.${system}.default = pkgs-stable.mkShell {
    packages = [
        pkgs-stable.haskellPackages.haskell-
language-server
        pkgs-stable.cabal-install
        pkgs-unstable.haskell.compiler.ghcHEAD
    ];
}
```

Figure 9. - A Nix expression for a dev shell exposing packages for the Haskell language sever, Haskell's compiler GHC and its build system cabal, as part of Flakes outputs.

The function `pkgs-stable.mkShell` is part of Nix's standard library. It a generalised form of the `stdenv.mkDerivation` function and abstracts away some of the common repetition of making shell environments. Other as part of a flake output, nix-shell environments can also be declared using a `shell.nix` file. Nix-shells provide a lightweight alternative to development environments compared to heavier handed Docker. Although there is less isolation of the resulting applications processes, nix-shells require much less resources.

VII. Open source software for the Arts

The appeal of open *Free and Open Source Software* (FOSS) is the increased control over the code running on their machine. As a package and easily be audited by reading through its source code, this level of transparency in turn, increases the security of FOSS software as bugs in the source code can be spotted and patched by that's software's community. FOSS software is very stable for long-term projects due to the distributed nature of the source code. If a FOSS reliant part of program or application grows stale or falls into disrepair by its original creators, then that part of the program can just as easily be continued by community at large and have development continued. Over the past decade, this kind of permissive licensing that allows for easy incorporation of FOSS projects has been promoted on source code hosting platforms such as Github [13].

Nix is fundamentally built on a source deployment model, where packages are built from source Nix deviations into its unique path in the Nix store. This is apposed to the majority of package managers in which the software distribution is binary based. For a lot of users, building from source can be frustrating, due the time it can take. However building from source with Nix allows for a high level of reproducibility, flexibility and transparency in open source software. By having the compilation of the software taking place on the end users personal hardware a large swath of supply chain vulnerabilities are avoided [2].

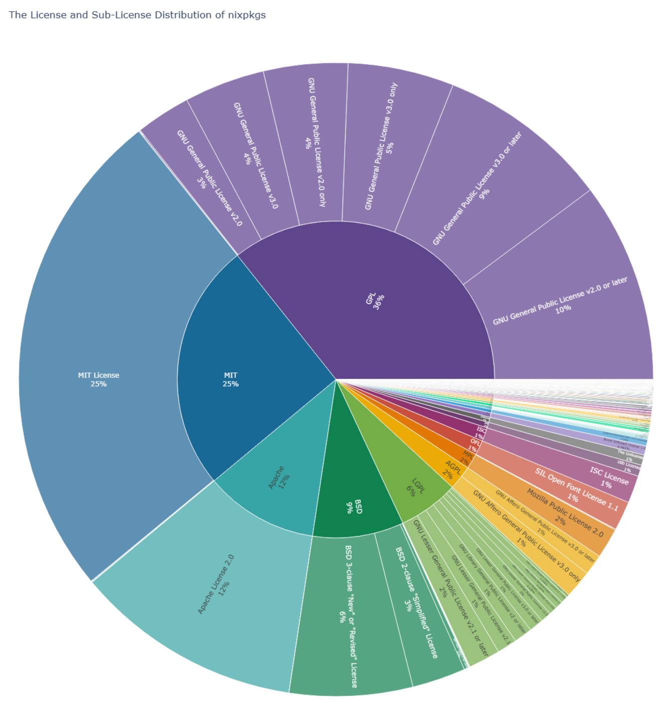

Figure 11. - The license and sub-license distribution in a sample of 16789 different packages available from the nixpkgs repository for the x86_64-linux platform.

VIII. Nixpkgs as a source of free and open source software

In this work we evaluated all 22,000 packages in available to the `x86_64-linux` platform to gleam their license metadata. We found that 5 main license types accounted for 87% for the license used in packages within the repository.

The first permissive license, Berkeley Software Distribution (BSD) , was created by University of California Berkeley. The BSD license allows for modification and distribution of the covered software. It comes in either a simplified 2 clause variant, where the non-endorsement clause is omitted or 3 clause variant which prevents the use of the original authors names or trademarks without explicit written permission. In our sample of the nixpkgs repository, 6% of packages use the BSD-3 clause as apposed to 3% using BSD-2. In total with 9%, the BSD license is 4th most used license.

Following the BSD license, the Massachusetts Institute of Technology (MIT) created an open source license based on the original BSD license. The main difference between the MIT and BSD licenses is that MIT does not contain any clauses with regards to promotion or advertising material, but does contain a attribution clause. It is most similar to the BSD-2 license. The MIT license also explicitly allows for the merging, publishing, sublicensing and selling of open source software under its license. 25% of the packages in nixpkgs fall under this license making it the largest single license type in the repository.

Approximately 12% of packages in the nixpkgs repository fall under the Apache License, it the most comprehensive and explicit of the permissive licenses and offers more legal advantages for simple licenses while providing similar grants. The Apache license contains a section on patents, including a patent retaliation clause amongst others.

The GNU General Public License (GPL) is the single largest type of license in the nixpkgs repository, being carried by 36% of the available packages in our sample. Its a reciprocal license that mandates that derivative works must release under a license offering the same freedoms as the GPL license. Like other FOSS licenses it allows for the modification and distribution of the source code. The Lesser General Public License (LGPL) constitutes 6% of the packages in nixpkgs.

With the majority of packages in nixpkgs being openly permissive through the MIT, Apache and BSL licenses (46% combined), licence compatibly of open source components to an artwork remains manageable and flexible with the permissive nature of these licences allowing the creator of the artwork the digression of how they want to handle the source code for their project. Copyleft license are the next largest group in the nixpkgs repository accounting for 42% of the packages in our sample, artworks which that would fall under this license, would have to provide their source code to each institution that hosts said work, however in efforts to reproduce and preserve digital work, supplying artwork with source may ultimately be preferred, especially when utilising the source deployment model which Nix thrives upon.

Copyleft Licenses Copyleft is an Intellectual Property legal technique which asserts that derivative works of a copyrighted material are persevered with same freedom to modify and distribute as its source material.

Our sample also contained a small amount closed source and proprietary licenses namely:

- Business Source License 1.1
- Server Side Public License
- Elastic License 2.0
- Functional Source License (all variants)
- Sustainable Use License
- Fraunhofer FDK AAC Codec Library License

- Creative Commons licenses with NC (NonCommercial) or ND (NoDerivatives) clauses
- Apple Public Source License 2.0

Which in total constituted less than 1% of total packages sampled from nixpkgs.

IX. Conclusion

In conclusion, Nix provides a powerful system for building software from source, with Nix flakes allowing for highly modular and bespoke build steps that can be tailored to each artwork that is built using the flake. Nix simplifies the build process for many programming languages allows for external dependencies for a given artwork to be cleanly packaged along with the source code as part of the packages build inputs. The reproducible binaries that Nix produces also ensure that the behaviour of artworks build from source using Nix, especially generative artworks, behaviour remains consistent between systems. The constitution of licenses in the nixpkgs repository naturally supports this, with 87% of packages using license which are either completely permissive or copyleft. In the aspiration of preservation for software based artworks, It may be prudent to follow the example of nixpkgs and utilise the GNU GPL licenses as they necessitate the sharing of source code upon distribution of the software.

As Nix flakes can expose other flakes, a common pattern is to aggregate multiple flakes together as flake inputs and re-exposing their outputs. This allows for a unified interface building any Nix packages or apps that a flake exposes, allowing for ergonomic access through via the command: `nix run .#packageName`, where everything before the hash denotes the aggregate flake's location, which in this case is the current directory (naturally it can also be remote) and the name of the desired package is specified after the hash. Since `nix run` only builds the binary specified in the flake and does not actually download the package, aggregate flakes in conjunction with nix run allows for Nix to act as an efficient form of internal distribution for artworks packaged via flakes between systems running Nix within an institution. Nix and Nix flakes can also serve as a means of preservation. By dutifully reproducing software based artworks with Nix, and maintaining the associated flakes which outline the required build steps, Nix can be used to generate identical instances of the artwork for many years to come.

Even if a given artwork has does not have its source code available, Nix can still provide many benefits, by leveraging the intricacies of the Nix Store, Nix can be used as a means for version controlling pre-compiled binaries. By making a derivation for each version of a binary, that is pinned by their respective version hashes, they will all receive their own unique Nix store paths allowing for multiple versions to co-exist at the same time within a system. Furthermore for artworks which require other packages to be on a system as a dependency, nix-shells provide a declarative and reproducible way generate the required environment for the artwork. These are both noted benefits which can advantage both artworks that are built from source and even ones in which the source code is not provided.

Both building from source with Nix and the use of reproducible environments, can serve not only to preserve artwork but also to reduce pain points in the setting up and distribution of software based artwork across different machines and institutions.

Data availability statement

All source data for this project can be found at the projects accompanying repository: https://github.com/ShilohAlleyne/nix-for-the-arts. As nixpkgs is a set of functions and derivations rather than a static database, to gain a package list for the x86_64-linux platform, the attribute-set containing each package name was first evaluated in the nix repl and each package name was then recorded as a JSON list. Each package was then evaluated to parse its licensing metadata. Each nix expression was ran programactally by calling it as an external process in Rust using `std::process::Command` and the metadata JSON was captured and parsed with the *serde* create. Visualisation was done in Python using the *plotly* library.

Acknowledgements

I would like to acknowledge the many valuable resources that the Nix committee and community have made freely available, showing contrary to popular opinion, that Nix is actually quite well documented. I would also like to pass my gratitude to Patricia Falcao, who was been a wonderful mentor to myself and to express thanks for the opportunity to write this paper.

Bibliography

[1] J. Hughes, "Why Functional Programming Matters," *The Computer Journal*, vol. 32, pp. 98–107, 1984.

[2] P. Dellaiera, "Reproducibility in Software Engineering," *Zenodo (CERN European Organization for Nuclear Research)*, Jun. 2024, doi: https://doi.org/10.5281/zenodo.12666899.

[3] M. Hausch, S. Hauser, and B. Uekermann, "Improving reproducibility of scientific software using Nix/NixOS: A case study on the preCICE ecosystem," *Electronic Communications of the EASST*, vol. 83, 2025, doi: 10.14279/eceasst.v83.2613.

[4] "Nix Language Basics — Nix.dev Documentation." [Online]. Available: https://nix.dev/tutorials/nix-language.html#overview

[5] E. Dolstra, "Maximal Laziness," *Electronic Notes in Theoretical Computer Science*, vol. 238, no. 5, pp. 81–99, Oct. 2009, doi: 10.1016/j.entcs.2009.09.042.

[6] E. Dolstra, M. de Jonge, and E. Visser, "Nix: A Safe and Policy-Free System for Software Deployment," in *Proceedings of the 18th USENIX Conference on System Administration*, in LISA '04. Atlanta, GA, USA: USENIX Association, 2004, pp. 79–92.

[7] A. Mokhov, N. Mitchell, and S. Peyton Jones, "Build Systems à la Carte: Theory and Practice," *Journal of Functional Programming*, vol. 30, Jan. 2020, doi: 10.1017/s0956796820000088.

[8] K. Claessen, "A Poor Man's Concurrency Monad," *Journal of Functional Programming*, vol. 9, no. 3, pp. 313–323, May 1999, doi: 10.1017/S0956796899003342.

[9] V. J. Illmer, ""Works on My Machine": A Case Study of Replicability Challenges in Computational Humanities Research," *Anthology of Computers and the Humanities*, vol. 3, no., pp. 142–148, Nov. 2025, doi: 10.63744/iaqeoznkfkuz.

[10] "Overlays - NixOS Wiki." [Online]. Available: https://nixos.wiki/wiki/Overlays

[11] Y. Gamage, D. Tiwari, M. Monperrus, and B. Baudry, "The Design Space of Lockfiles Across Package Managers," *arXiv.org*, 2025, doi: 10.48550/arXiv.2505.04834.

[12] B. Rad, H. Bhatti, and M. Ahmadi, "An Introduction to Docker and Analysis of its Performance," *IJCSNS International Journal*

of Computer Science and Network Security, vol. 17, no. 3, pp. 228–233, 2017, [Online]. Available: http://paper.ijcsns.org/07_book/201703/20170327.pdf

[13] P. Pícha and S. Serbout, "On the Adoption of Open Source Software Licensing – A Pattern Collection," in *Proceedings of the 29th European Conference on Pattern Languages of Programs, People, and Practices*, in EuroPLoP '24. New York, NY, USA: Association for Computing Machinery, 2024. doi: 10.1145/3698322.3698341.